EXTENDING THE TIME-STAMP ORDERING TECHNIQUE IN SQLITE AND PERFORMING COMPARATIVE ANALYSIS.

Oohasripriya Nandipati
Data Science and Analytics
University of Oklahoma
Norman, Oklahoma, USA
oohasripriya.nandipati-1@ou.edu

Mounica Pragyna Ravilla
Data Science and Analytics
University of Oklahoma
Norman, Oklahoma, USA
mounica.pragyna.ravilla-1@ou.edu

Farhan Hussain Abedi Syed
Computer Science Department
University of Oklahoma
Edmond, Oklahoma, USA
Farhan.Hussain.Abedi.Syed-1@ou.edu

ABSTRACT

Concurrency control is an integral part of database. This ensures the correctness of databases when transactions are processed in parallel. Given the ever-growing number of concurrency control algorithms, the database designer is faced with the difficult decision of choosing the best algorithm for an application. In this paper we propose an implementation of an algorithm for the basic timestamp-ordering concurrency control in centralized database system (SQLite) to improve the degree of concurrency. Currently, SQLite is using two-phase locking system [7]. This paper conducts comparative simulation experiments between Hctree, a specialized backend for SQLite, and SQLite itself, evaluating their respective performances alongside the timestamp ordering algorithm integrated within SQLite.

KEYWORDS

Concurrency control, timestamp, SQLite, Two-phase Locking, Database Consistency.

ACM Reference format:

Orji, C.U., Lilien, L., & Hyriak, J. (n.d.). A Performance Analysis of Optimistic and Centralized Database Systems: A Basic Timestamp-Ordering Concurrency Control Algorithm. Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, Illinois 60880.

1 INTRODUCTION

Transaction Processing has been used in many situations such as for bank transfers and credit card payments, and in distributed file systems for big science. The rise of many-core architecture has led to a growing interest in faster concurrent transaction processing. It is necessary to ensure the correctness (i.e., serializability) of the resulting database after the execution of the transactions. Concurrency control takes this role, ensuring the isolation of transactions. When two or more transactions execute concurrently, their database operations execute in an interleaved fashion. That is, operations from one transaction may execute in between two operations from another transaction. This interleaving can cause transactions to produce incorrect results, thereby leaving the database in an inconsistent state [5].

Concurrency control is mainly divided into two categories: pessimistic and optimistic. One of the most scalable protocols is the two-phase locking protocol (2PL), which is pessimistic and suited for contended workloads [1].

This time-stamp implementation is important as SQLite works fine if there are not more than about 50 users working towards the database concurrently (though depending on whether it is reads or writes). Once there are more than this, they will notice a slowdown if there are a lot of concurrent writing on the server as lots of time is spent on locks, and there is nothing like a cache as there is no database server.

Hctree is designed with high focus on concurrency and replication scenarios. In this paper we focus on concurrency. SQLite permits only a single concurrent writer, while HC-tree supports dozens of concurrent writers with optimistic row-level locking, markedly enhancing concurrency compared to SQLite's default behavior. In this paper we present a framework for the design and analysis of concurrency control algorithms for Centralized database management systems (CDBMS). We'll be comparing Hctree, SQLite, and integrated timestamp method [8].

2 RELATED WORKS

Below are some of the related works found through Literatures.

2.1 Performance Analysis of an Optimistic and for Centralized Database Systems a basic Timestamp-Ordering Concurrency Control Algorithms: In this paper they have discussed briefly about optimistic concurrency control algorithm and Timestamp and compared which one performs better in Centralized Database System and concluded that optimistic approach is better for transaction mizes dominated by retrievals. For transaction mizes dominated by updates, the optimistic algorithm spends time performing operations that have a good chance of being voided by earlier conflicting operations.

This paper is helpful to undergo details about how timestamp algorithm works in Centralized Database. Our project aims to enhance concurrency control in SQLite, specifically by improving upon the traditional two-phase locking system.

2.2 Scalable Timestamp Allocation for Deadlock Prevention in Two-phase Locking based Protocols: Bamboo is a concurrency control protocol, an extension of 2PL Wound-Wait **(**wounding a younger transaction if it tries to lock a resource held by an older transaction**).** One problem of Bamboo is that it requires transactions to fetch timestamps from a single centralized atomic counter. This literature proposes three decentralization methods to address this problem**.** Thread-ID (TID): Assigns a unique ID to each thread, allowing transactions to use these IDs as timestamps, FairTID: An extension of TID that maintains performance even in high-contention settings, RandID: Allocates timestamps using random number generators. results show up to 62% and 59% throughput improvement with FairTID and RandID, respectively, compared to Bamboo.

This paper **is** an alternative approach to our project, as both aim to enhance throughput performance by addressing deadlocks in the two-phase locking (2PL) mechanism. However, while the paper focuses on optimizing 2PL, our work adopts a different strategy by utilizing timestamp ordering instead of 2PL. Additionally, we conducted a comparative analysis between SQLite and HCTree**,** this makes our project different from the referenced work.

3 PROPOSED WORKS:

3.1 CONCURRENCY CONTROL IN CENTRALIZED DATABASE:

A centralized database system is when there's only one main copy of the database that all users share and access locally [4]. Users interact with the database by issuing transactions**.** These transactions can be either asking the database for information (like a search) or giving it instructions (like updating data). We assume that each transaction is complete and correct by itself. This means that if a transaction runs by itself, it will finish without any problems, and it won't mess up any rules that the database has in place for the data it works with. Each transaction may have a read-set, which is the set of data items it reads, and a write-set, which is the set of data items it writes. Two transactions are said to be in conflict if the intersection of the write-set of one with the read-set (or write-set) of the other is not null. Concurrency of transactions refers to the execution of more than one transaction at the same time. More precisely, it refers to the execution of one transaction before the completion of other outstanding transactions. If concurrent transactions do not maintain inter-consistency over the database, two important anomalies can occur. **Lost Update:** This happens when one transaction updates a piece of data, but before it's done, another transaction updates the same data, causing the first update to be lost or overwritten**, Inconsistent Retrievals:** This occurs when different transactions read the same data at different times, but get different results because the data has been changed by other transactions in between [9].

3.2 TIMESTAMP ALGORITHM:

One of the solutions for concurrency control problems in centralized database is timestamp. The database operations we use in transactions are tread(i,n) is invoked by a transaction to return the value of the object number i in the local variable n, twrite(n,i) is invoked by a transaction that wishes to change the value of the object number i to the value of the variable n which is a local copy of the object.[5]. A centralized DBMS consists of one TM and one DM executing at the same site [6]. A transaction, T, accesses the DBMS by issuing BEGIN, READ, WRITE, and END operations. In timestamp transactions are serialized based on predefined timestamps [10]. When a transaction begins, a unique timestamp is generated by the transaction manager (TM) using the local clock time, ensuring system-wide uniqueness. Transactions attach these timestamps to all read and write operations, with conflicting operations processed in timestamp order. For read-write (rw) synchronization, conflicts arise if both operations affect the same data item, with one being a read and the other a write. Similarly, for write-write (ww) synchronization, conflicts occur when both operations are writes on the same data item. T/O ensures acyclic relations for rw and ww synchronizations, facilitating valid serialization orders. Implementation involves building a scheduler that tracks timestamps for data items and processes operations accordingly. For rw synchronization, the scheduler compares timestamps to determine operation validity, updating timestamps as necessary. Similarly, for ww synchronization, the scheduler evaluates timestamps to accept or reject operations. Transaction aborts result in timestamp updates to prevent cyclic restarts, albeit requiring substantial storage for timestamp maintenance, with potential optimizations discussed for reducing storage overhead.

3.3 SYSTEM DIAGRAM OF SQLITE:

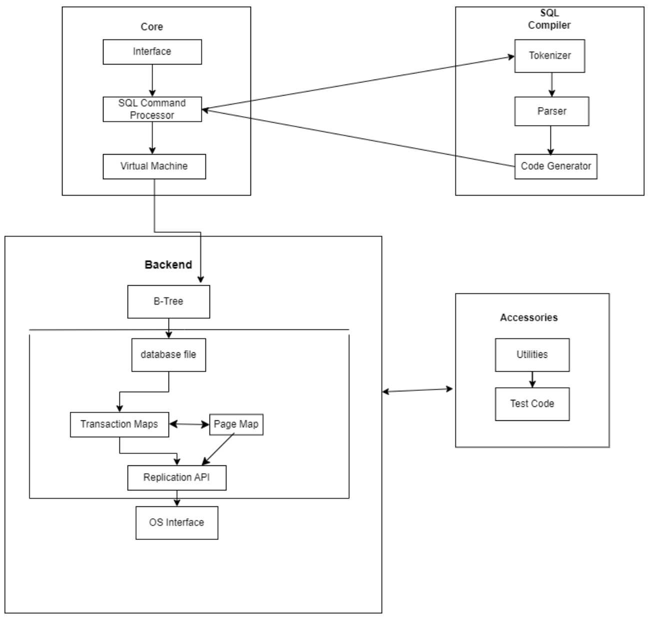


Figure 1: Structure of SQLite.

3.3.1 Current concurrency control mechanism in Hctree:

The concurrency lies between Transaction Maps and Page Map.

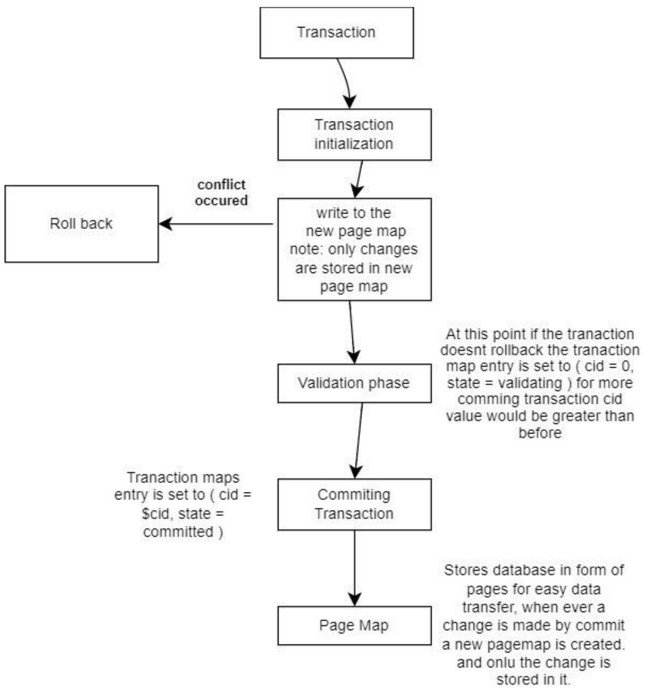

Transaction
Transaction initialization
conflict occured
Roll back
write to the new page map note: only changes are stored in new page map
Validation phase
At this point if the tranaction doesnt rollback the tranaction map entry is set to (cid = 0, state = validating) for more comming transaction cid value would be greater than before
Commiting Transaction
Tranaction maps entry is set to (cid = $cid, state = committed)
Page Map
Stores database in form of pages for easy data transfer, when ever a change is made by commit a new pagemap is created. and onlu the change is stored in it.

Figure 2: Abou Transaction and Page Map in SQLite.

The core components of the system include an interface for SQL commands, a command processor, SQL compiler with tokenizer and parser, and a code generator utilizing a virtual machine. Backend operations involve the Btree structure for data storage, management of database files, and Transaction Maps for concurrency control implementation. Transaction Maps assign unique IDs to transactions and track their status, employing optimistic concurrency control based on keys and logical key ranges to prevent conflicts. Conflict detection relies on snapshots of the database, rolling back transactions if conflicts arise. During validation, transactions are assigned commit IDs, and any conflicting changes are rolled back before committing. Page maps store database changes for easy retrieval, while the Replication API facilitates distributed environments by replicating changes across connected nodes. The OS Interface provides essential functions for file management and system interaction.

The **Transaction Maps** component serves as a pivotal storehouse for incoming transactions within the system, assigning a unique Transaction ID (TID) to each and tracking their status, whether they are in progress, ready to commit, failed, or committed. Additionally, it allocates a **Commit ID** (CID) for successfully committed transactions. Concurrency control mechanisms are embedded within Transaction Maps, employing an optimistic approach based on keys and logical key ranges rather than page sets. This design ensures that conflicts arise only when transactions attempt to modify closely situated values within the same tables or indexes, avoiding conflicts between transactions operating on distinct sets of tables. **Transaction Initialization** involves assigning a TID upon transaction initiation, enabling the system to correlate database changes with respective transactions. Conflict Detection relies on snapshots, providing a read-only view of the database at transaction initiation. If a transaction encounters a version of a key with a TID indicating modification after its snapshot, it's flagged as a conflict, prompting rollback to maintain database consistency. During the **Validation Phase**, transactions are assigned CIDs, and conflicts are detected by comparing TIDs against the database snapshot CID. **Committing Transactions** updates the Transaction Maps state to reflect the transaction's commitment status. The Page maps store database changes in page format, with new changes creating updated page maps while older versions are retained for historical reference. In distributed environments, the Replication API facilitates the dissemination of changes across nodes, enabling synchronization of replicated databases. The **OS Interface** provides essential file management and system interaction functionalities, ensuring seamless operation of the system.

3.3.2 QUERING MODEL FOR THE TIMESTAMP:

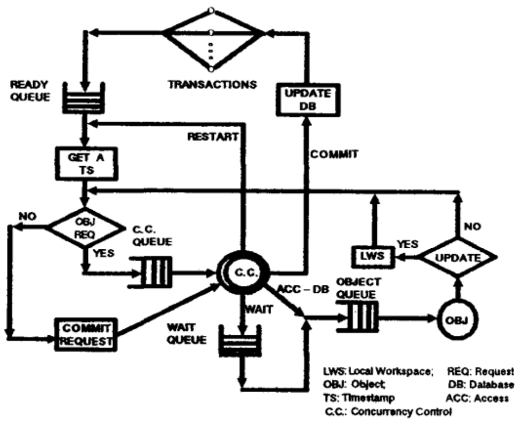


Figure 3: Logical Quering Model for the Timestamp [5]

The transaction processing system begins with transactions entering the system, marking the starting point of the process. Transactions then enter the READY QUEUE, signaling their readiness for processing, and each transaction requests a timestamp to establish its order of execution. Following this, transactions proceed to request access to specific database objects through OBJ REQ for read or write operations. Subsequently, they enter the Concurrency Control Queue (C.C. QUEUE) to manage concurrent access to database objects. The Concurrency Control module (C.C.) ensures transactions access database objects in a serializable manner, maintaining data integrity and preventing conflicts. Upon being granted access to the database (ACC-DB), transactions operate within their Local Workspace (LWS) to perform read or write operations, subsequently updating the database. If transactions complete successfully, they commit their changes to the database; however, in cases of conflicts or issues, they may need to be restarted. Object Queue manages access to specific database objects orderly, while the WAIT QUEUE accommodates transactions waiting for object access. Throughout the process, decisions such as commit requests and conflict resolutions are made at various points, dictating the flow of transactions based on conditions encountered.

3.4 IMPLEMENTATION OF TIMESTAMP CONCURRENCY CONTROL IN SQLITE

The proposed work integrates a timestamp-based concurrency control system into SQLite to address the bottlenecks faced by its traditional two-phase locking mechanism, especially under high concurrent write scenarios. This integration, while maintaining SQLite's lightweight footprint, introduces a novel concurrency control method to improve the degree of concurrency and system throughput.

3.4.1 Timestamp-Based Concurrency Control Mechanism

In our approach, each transaction receives a unique timestamp, serving as a serial ordering mechanism. These timestamps dictate the execution sequence of transactions, ensuring a conflict-free and consistent state of the database. The enhancements to SQLite's architecture involve the following modifications:

The sqlite3 structure is expanded to include a transactionTimestamp field to manage concurrency timestamps. The Timestamps are initialized at the beginning of a transaction, employing the sqlite3BeginTransaction function to associate a unique timestamp to each transaction based on the system clock. Modifications to data are managed by checking the transaction's timestamp against the data's last modified timestamp. This check is integrated into data mutation functions such as sqlite3BtreeInsert. Functions to generate and compare timestamps, such as get_current_timestamp and can_proceed, are implemented to support timestamp logic. The sqlite3CommitTransaction and sqlite3RollbackTransaction functions incorporate logic to finalize the timestamps or handle conflicts as required.

Timestamp Pseudocode

```
/* Extend the sqlite3 structure to include transaction timestamp
*/
typedef struct sqlite3 {
  /* Other existing fields */
  int transactionTimestamp; // Timestamp for concurrency control
} sqlite3;
/* Utility function to get the current system time as a timestamp
*/
int get_current_timestamp() {
  return (int)time(NULL); // Simplified version, normally you might
want more resolution
}
/* Function to check if the transaction can proceed based on
timestamps */
int can_proceed(sqlite3 *db, int lastModifiedTimestamp) {
  return db->transactionTimestamp > lastModifiedTimestamp;
}
/* Function to initialize a transaction, setting up its timestamp */
void begin_transaction(sqlite3 *db) {
  db->transactionTimestamp = get_current_timestamp();
}
```

The sqlite3 data structure, which is central to the SQLite database connection and transaction management, is extended to include an additional integer field named transactionTimestamp. This new field is critical for tracking the timing of each transaction within the concurrency control mechanism. [3]

In the beginning phase of transaction processing, the function sqlite3BeginTransaction is responsible for initializing the transaction timestamp. Upon invocation, it calls the get_current_timestamp function to retrieve the current system time, represented as an integer, and assigns this value to the transactionTimestamp field of the sqlite3 structure. This timestamp will serve as a unique identifier to maintain the chronological order of transactions and is essential for the concurrency control logic.

The get_current_timestamp function is designed to interact with the system clock to fetch the current time, which is then used to generate a unique timestamp for the transaction. It ensures that each transaction is tagged with a specific point in time, providing the ability to sort transactions in a sequential order based on their start times.

A key function in the concurrency control mechanism is can_proceed, which takes the database connection and the last modified timestamp of the data item as inputs. This function compares the transactionTimestamp with the lastModifiedTimestamp, allowing the transaction to proceed only if its timestamp is later than the last modified timestamp, indicating that no other committed transaction has modified the data item since this transaction began. If this check fails, the function advises against proceeding, thus preventing conflicts that could lead to inconsistencies within the database.

The modify_data function encapsulates the logic for updating data within the database. Before any data modification takes place, it invokes can_proceed to determine whether the transaction's timestamp permits the modification. If the check passes, the function carries out the necessary data manipulation and signals a successful operation. If the check fails, the function aborts the transaction, signaling a conflict due to the timestamp check, and no changes are made to the database.

When a transaction reaches its end and is ready to commit, the sqlite3CommitTransaction function is called. This function includes additional logic to validate the transaction's timestamp against the current state of the database. It ensures that no intervening transactions have caused a state change that would invalidate the current transaction. If the final timestamp checks are successful, the transaction is committed; otherwise, it is aborted.

In cases where a transaction cannot be successfully committed, the sqlite3RollbackTransaction function is invoked to undo any changes that the transaction may have made. This function also handles any updates or resets to the timestamps to reflect the transaction's rollback, ensuring that the database remains in a consistent state even after a transaction failure.

Lastly, when inserting new data into the database, the function sqlite3BtreeInsert is modified to include a timestamp verification step. It uses the can proceed function to ensure that the insertion does not violate the chronological order of transactions as determined by their timestamps. If a timestamp conflict is detected, the function returns an error code indicating the conflict, preventing the insertion from proceeding and maintaining the integrity of the database state.

Overall, the introduction of timestamp-based concurrency control requires precise adjustments to the SQLite source code, ensuring that all transaction operations adhere to the new temporal logic that governs transaction ordering and consistency. This approach aims to improve the database's performance and its ability to handle concurrent transactions without incurring the overhead and complexity of traditional locking mechanisms.[2]

3.4.2 Timestamp-Based Concurrency Control Testing and Analysis

To evaluate the effectiveness of the newly implemented timestamp concurrency control in SQLite, a comprehensive testing methodology is employed, involving multiple threads executing both read and write operations. This testing approach aims to ensure that the timestamp-based concurrency mechanism adheres to the expected behavior under different transaction scenarios, particularly focusing on the system's ability to handle concurrent operations without data inconsistency.

The first step involves compiling the modified SQLite source code that integrates the timestamp concurrency control. This compilation is done using Visual Studio Code's integrated terminal, ensuring that the new code changes are active. A new SQLite database is then created, which includes nine tables named from t1 to t9, each containing two attributes: id and name. These tables are populated with preliminary data to facilitate varied transaction operations during the testing phase.

SQLite Command-line shell

```
.open test.db
CREATE TABLE t1 (id INTEGER PRIMARY KEY, name TEXT);
```

Transactions for testing

```
T0: SELECT * FROM t1 WHERE id='12';
T1: SELECT id, name FROM t1 WHERE id='12';
T2: UPDATE t1 SET name='aa1' WHERE id='11';
T3: UPDATE t3 SET name='cc1' WHERE id='31';
T4: SELECT * FROM t4 WHERE id='41';
```

*T5: SELECT * FROM t5 WHERE id='51';*

T6: UPDATE t6 SET name='hh1' WHERE id='61';

T7: UPDATE t7 SET name='gg1' WHERE id='71';

T8: UPDATE t7 SET name='gg1' WHERE id='72';

T9: UPDATE t9 SET name='ii1' WHERE id='91';

The core of the testing framework is a multi-threaded application, designed to simulate concurrent database interactions. This application opens multiple connections to the SQLite database and creates numerous threads, each assigned specific read and write operations. These operations are mapped out to mimic realistic transaction patterns, as identified in the preliminary design phase. Transactions from T0 to T9 are distributed among these threads, with careful attention to the timing and sequence of operations to trigger potential concurrency conflicts.

Python is chosen for implementing the test due to its robust threading support and ease of database interaction via the *sqlite3* module. The test code includes functions to execute SQL commands, handle transactions, and log the execution time for each operation. Threads are used to initiate database transactions, where read operations are expected to proceed without hindrance, and write operations are scrutinized for conflicts against ongoing reads or other writes. The transactions are executed in two phases, Time1 and Time2, to distinguish between concurrent reads and the interaction between reads and writes.

The *execute_transaction* function handles database operations. Each thread calls this function to execute a specified SQL command on a database that is stored entirely in memory *(mode=memory)*. This in-memory database allows for fast access and changes without any permanent disk writes.

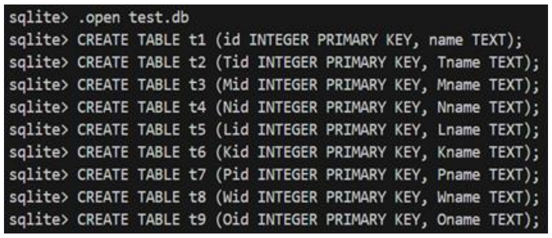
The script sets up a connection to the SQLite database and creates a cursor for executing SQL commands. It measures the time taken to execute each command, providing insights into the efficiency of the database operations under the concurrency control system.

For *SELECT* queries (read operations), the results are fetched and printed along with the transaction name. For other types of queries (like *UPDATE*, *INSERT*), the transaction is committed to the database.

Any exceptions (errors) encountered during the execution of a command are caught and reported, indicating whether a transaction failed and why. Transactions are defined in a dictionary called transactions, where each transaction is labeled (e.g., "T0", "T1") and associated with a specific SQL command.

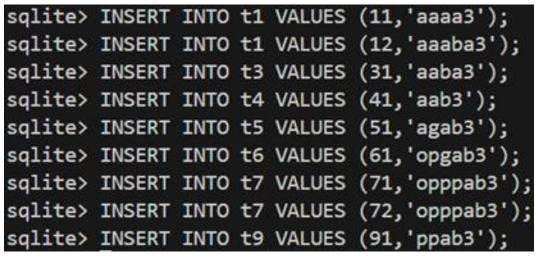
Threads are created for each transaction, allowing multiple operations to be performed concurrently, which is crucial for testing the concurrency control mechanisms of the database. The script assumes that transactions labeled from "T0" to "T4" are grouped under "Time1" and the rest under "Time2". This likely represents different phases or waves of transactions being tested to observe how read and write locks interact and are managed.

Threads for the "Time1" transactions are started first, with a slight delay (time.sleep(0.1)) between each start to stagger their execution slightly. This simulates a real-world scenario where client requests might not hit the server exactly at the same time. After all "Time1" threads are complete, "Time2" threads begin, following the same staggered start pattern.



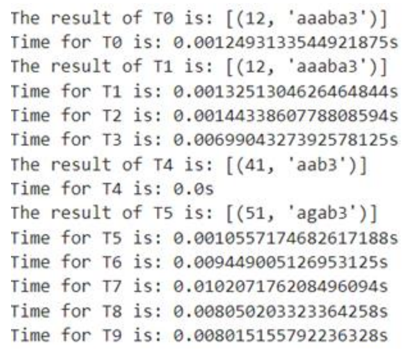
```
sqlite> .open test.db
sqlite> CREATE TABLE t1 (id INTEGER PRIMARY KEY, name TEXT);
sqlite> CREATE TABLE t2 (Tid INTEGER PRIMARY KEY, Tname TEXT);
sqlite> CREATE TABLE t3 (Mid INTEGER PRIMARY KEY, Mname TEXT);
sqlite> CREATE TABLE t4 (Nid INTEGER PRIMARY KEY, Nname TEXT);
sqlite> CREATE TABLE t5 (Lid INTEGER PRIMARY KEY, Lname TEXT);
sqlite> CREATE TABLE t6 (Kid INTEGER PRIMARY KEY, Kname TEXT);
sqlite> CREATE TABLE t7 (Pid INTEGER PRIMARY KEY, Pname TEXT);
sqlite> CREATE TABLE t8 (Wid INTEGER PRIMARY KEY, Wname TEXT);
sqlite> CREATE TABLE t9 (Oid INTEGER PRIMARY KEY, Oname TEXT);
```

Figure 4. SQLite command lines for creating database/tables



```
sqlite> INSERT INTO t1 VALUES (11,'aaaa3');
sqlite> INSERT INTO t1 VALUES (12,'aaaba3');
sqlite> INSERT INTO t3 VALUES (31,'aaba3');
sqlite> INSERT INTO t4 VALUES (41,'aab3');
sqlite> INSERT INTO t5 VALUES (51,'agab3');
sqlite> INSERT INTO t6 VALUES (61,'opgab3');
sqlite> INSERT INTO t7 VALUES (71,'opppab3');
sqlite> INSERT INTO t7 VALUES (72,'opppab3');
sqlite> INSERT INTO t9 VALUES (91,'ppab3');
```

Figure 5. SQLite commands showing insertion into tables



```
The result of T0 is: [(12, 'aaaba3')]
Time for T0 is: 0.0012493133544921875s
The result of T1 is: [(12, 'aaaba3')]
Time for T1 is: 0.001325130462646484s
Time for T2 is: 0.001443386077880859s
Time for T3 is: 0.006990432739257812s
The result of T4 is: [(41, 'aab3')]
Time for T4 is: 0.0s
The result of T5 is: [(51, 'agab3')]
Time for T5 is: 0.001055717468261718s
Time for T6 is: 0.009449005126953125s
Time for T7 is: 0.010207176208496094s
Time for T8 is: 0.008050203323364258s
Time for T9 is: 0.008015155792236328s
```

Figure 6. Execution of timestamp concurrency control

4.1 Multithreaded Database Operations with SQLite3

In our study, we developed a script specifically designed to test database systems like SQLite. Given that our Timestamp Implemented SQLite and Hctree share the same code structure, we utilized the same script, mptest.c, across all systems. This script is particularly useful for testing the concurrent access capabilities of a database in a multi-process or multi-threaded environment.

The mptest.c script operates by accepting a database name and a test script as arguments. We configured the system to perform the following tasks:

Task 1: Utilizes persistent journal mode and disables memory mapped I/O.

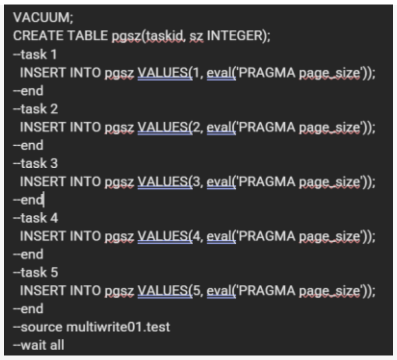
Task 2: Employs truncate journal mode and a small memory mapped I/O.

Task 3: Stores the rollback journal in RAM.

Task 4: Disables the rollback journal.

Task 5: Uses a large memory mapped I/O.

For each configuration, we executed five subtasks.



```
VACUUM;
CREATE TABLE pgsz(taskid, sz INTEGER);
--task 1
  INSERT INTO pgsz VALUES(1, eval('PRAGMA page_size'));
--end
--task 2
  INSERT INTO pgsz VALUES(2, eval('PRAGMA page_size'));
--end
--task 3
  INSERT INTO pgsz VALUES(3, eval('PRAGMA page_size'));
--end
--task 4
  INSERT INTO pgsz VALUES(4, eval('PRAGMA page_size'));
--end
--task 5
  INSERT INTO pgsz VALUES(5, eval('PRAGMA page_size'));
--end
--source multiwrite01.test
--wait all
```

Figure 7. Configuring subtasks for concurrency tests

4.1.1 Outcome

Our performance comparison revealed that all three configurations: Stock SQLite, Timestamp Implemented SQLite, and Hctree, exhibited near-zero errors, indicating a high level of reliability. Hctree led in the number of tests conducted, suggesting extensive testing. While Hctree had the longest execution time, which could be a factor for real-time applications where speed is paramount, it compensated with the highest throughput. This suggests that Hctree can handle a larger volume of data over time compared to the other two configurations.

This analysis offers valuable insights for selecting an appropriate database configuration based on specific application requirements such as speed, data volume, and reliability. It's crucial to note that the optimal choice would depend on the specific use case and requirements of the application. For instance, if speed is a priority, Stock SQLite might be the best choice, but if handling a large volume of data is more important, Hctree could be the preferred option.

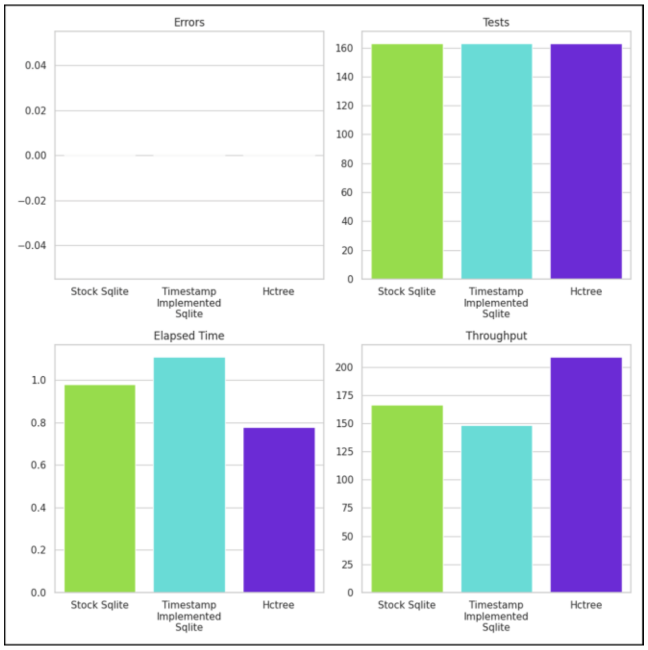


Figure 8: Results of Concurrency test.

4.2 Multithreaded Database Query Performance Testing:

In Section 4.2 of our study, we conducted Multithreaded Database Query Performance Testing on three different database systems: Stock SQLite, Timestamp Implemented SQLite, and Hctree. The goal was to evaluate the execution time for both read and write queries across these systems.

The following graphs are the results of each system that has been tested

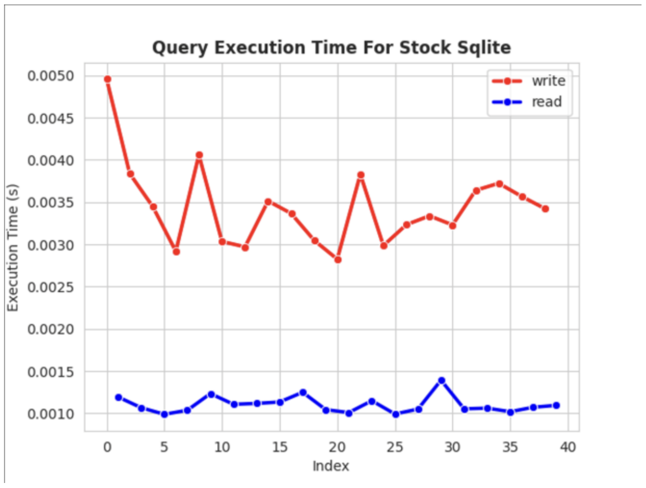


Figure 9: Read and Write Results of Stock sqlite Query execution performance test.

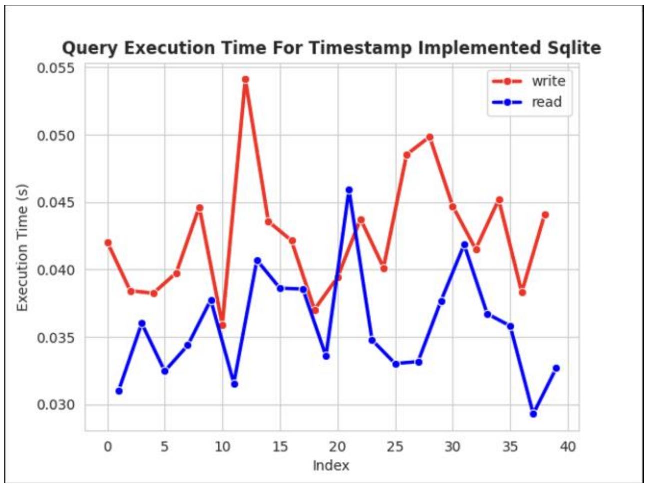


Figure 10: Read and Write Results of Timestamp Implemented Query execution performance test.

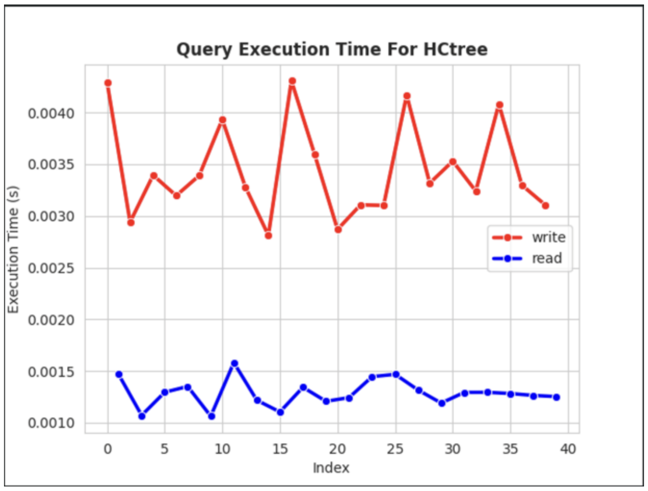


Figure 11: Read and Write results of HCtree Query execution performance Test.

The data reveals that Hctree consistently outperforms both Stock SQLitFute and Timestamp Implemented Sqlite in terms of execution time for both read and write queries. This suggests that Hctree is more efficient in handling both types of queries, making it a potentially preferable choice for applications that require high-speed read and write operations.

And the comparison for all the systems is as follows:

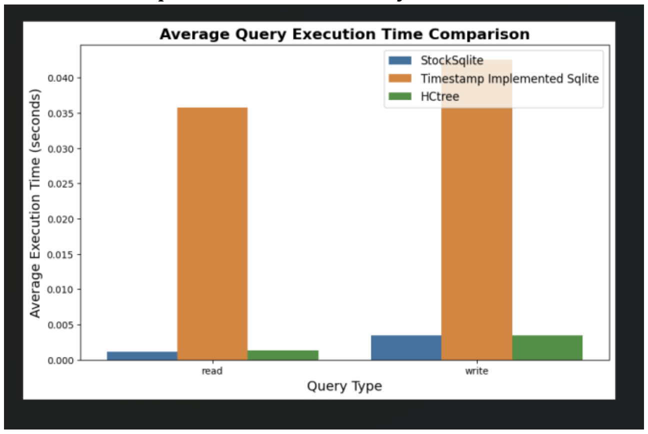


Figure 12: comparing the three system's Stock sqlite, Timestamp Implemented Sqlite, HCtree.

While our Timestamp Implemented SQLite does take a significant amount of time compared to Stock SQLite and Hctree, it's crucial to highlight that it may still maintain the same level of concurrency as demonstrated in the tests presented in Section 4.1. This suggests that despite the longer execution times, our implementation could effectively handle multiple concurrent operations, a key aspect in many real-world applications.

5. Conclusion:

In this paper we have extended SQLite timestamp to improve the concurrency and performed analysis on HCtree , SQLite. Our findings indicate that HCtree consistently outperforms both Stock SQLite and Timestamp Implemented SQLite in terms of execution time for both read and write operations.

Although the implementation of Timestamp protocol did almost have the same concurrency, the execution times slowed down. this study set out to achieve several objectives aimed at improving concurrency levels in SQLite through the implementation of a timestamp-based concurrency control algorithm.

6. FUTURE WORK:

While this study has demonstrated the effectiveness of implementing a timestamp-based concurrency control algorithm in SQLite to improve concurrency levels. Our future research is the investigation of alternative timestamp-based concurrency control techniques, such as the two-version timestamp approach. The two-version timestamp approach, which maintains separate timestamps for read and write operations, has shown promise in achieving high concurrency levels in database systems. Building upon the insights gained from our implementation of the basic timestamp ordering algorithm, future work could involve integrating and evaluating the performance of the two-version timestamp approach within SQLite. By conducting comparative simulation experiments like those described in this paper, we can assess the concurrency and scalability benefits of the two-version timestamp approach and compare its performance against other concurrency control mechanisms.

REFERENCES

[1] Nakamori, T., Nemoto, J., Hoshino, T., & Kawashima, H.(2023). Scalable Timestamp Allocation for Deadlock Prevention in Two-phase Locking based Protocols. Journal of Information Processing, 31(0), 365-374. https://doi.org/10.2197/ipsjjip.31.365

[2] Gaffney Kevin, P., Claus Robert, & Patel Jignesh, M. (2021). Database isolation by scheduling. Proceedings of the VLDB Endowment, 14(9), 1467-1480. https://doi.org/10.14778/3461535.3461537.

[3] Zhao, X., Ding, 1., Ma, R., Gong, Q., & Yang, Y. (2017). Research and improvement of SQLite's Concurrency Control Mechanism. 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering. 10.2991/icmmcce-17.2017.272.

[4] N. Kaur, R. Singh, A.K. Sarje, M. Misra, "Performance evaluation of secure concurrency control algorithm for multilevel secure distributed database system", International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II, vol.1, pp.249-254 Vol. 1, 2005.

[5] C. U. Orji, L. Lilien and J. Hyziak, "A performance analysis of an optimistic and a basic timestamp-ordering concurrency control algorithms for centralized database systems," Proceedings. Fourth International Conference on Data Engineering, Los Angeles, CA, USA, 1988, pp. 64-71, Doi: 10.1109/ICDE.1988.105447.

[6] Bernstein, P. A., & Goodman, N. TIMESTAMP-BASED ALGORITHMS FOR CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS. ,1980, Computer Corporation of America and Harvard University, IEEE.

[7] Architecture of SQLite. (n.d.). Retrieved March 25, 2024, from https://www.sqlite.org/arch.html

[8] hctree: Documentation. (n.d.). Retrieved March 25, 2024, from https://sqlite.org/hctree/doc/hctree/doc/hctree/design.wiki.

[9] Hsiao, D.K., & Ozsu, T.M. (1981). A Survey of Concurrency Control Mechanisms for Centralized and Distributed Databases. Computer and Information Science Research Center, The Ohio State University, Columbus, OH 43210. (February 1981).

[10] Aspnes, J., Fekete, A., Lynch, N., Merritt, M., & Weihl, W. (1992). A Theory of Timestamp-Based Concurrency Control for Nested Transactions. ACM Transactions on Database Systems (TODS), 17(1), 1-47.